Ex. 1, Audio

Stefan Karlsson

November 12, 2014

1 Our first examples: Sound

Sound is a long 1D signal and it is common with very large vectors in audio processing. Lets gather some audio data into our matlab workspace and play it[1]:

```
load handel;                          %loads new audio vector "y", and variable "Fs"
yHandel = y;

%first way to play audio:
sound(yHandel ,Fs);                   %single function

pause(9);                             %pause 9 seconds while playing

%second way to play audio:
AP = audioplayer(yHandel, Fs);        %create an audioplayer object first
play(AP);                             %play audio
```

For good quality sound, we need to have many samples, sometimes as much as 50 000 per second (`Fs`=50 kHz). Having 1 second of audio at 50 kHz results in a vector that is 50 000 elements long. We can generate audio synthetically as a single sinus wave:

```
Fs = 10000;                           %sampling frequency
k  = 4000;                            %audio frequency
t  = linspace(0,2,Fs*2);              %2 seconds long vector
y  = sin(t*k);                        %generate audio
```

A common mistake is to confuse the sampling frequency with the audio frequency. The sampling frequency tells us how many samples we have per second, and is a measure of the quality of the sound. The audio frequency tells us the pitch, or the tone of the audio. Very different indeed!

Lets take a look at the signal we synthesized:

```
plot(t(1:200), y(1:200));             %plot the first 200 samples
ylim([-3,3]);                         %set limits of the y-axis
title('pure frequency audio');
```

[1] In this example we will load some standard Matlab sounds that are available. If you want to load your own audio from file, use the function 'wavread'

We can change the intensity, the tone (frequency) and just about anything we like this way:

```
k = 6000;                                          %base tone

      %Make vectors of different tones and duration
y1 = sin( t(1:4000)*k     );
y2 = sin( t(1:4500)*k*1.12);
y3 = sin( t(1:5500)*k*0.89);
y4 = sin( t(1:4500)*k*0.45);
y5 = sin( t(1:6500)*k*0.67);

      %play some music:
sound([y1 y2 y3 y4 y5],Fs);
```

We are in this example dealing with two signals, at two different scales: on the fines scale, we define audio samples. These are actual values of air density offsets, as produced by a speaker system. The coarsest scale we deal with, is the notation `[y1 y2 y3 y4 y5]`. We will dive into a few intermediate steps in this exercise, and synthesize music. This is a nice example where you will work with everything from the finest, to scales dealing with audio effects, up to finally the scale of musical notation.

2 Audio Synthesis

Provided with this pdf, you have a function file: 'prettyNote1.m', that is designed to synthesize notes. Lets investigate the function `prettyNote1` starting with the syntax:

```
y = prettyNote1(TheNote,bPlay,bDisplay)        %syntax for the function
```

Here is a specification of `prettyNote1`:

- `y`(Audio Vector), synthesized audio.
- `TheNote` (2D vector), information about the note to be synthesized:
  - `TheNote(1)`: The frequency of the tone.
  - `TheNote(2)`: The duration(in seconds) of the sound.
- `bPlay` (boolean[2], default: false), use to playback sound.
- `bDisplay` (boolean, default: false), use to plot audio vectors.

We will have different versions (`prettyNote1` up to `prettyNote4)`.They will share the same syntax, but generate different kinds of sound. The sound of `prettyNote1` is the pure tone, given by:

$$y = \sin(tP)$$

where P is the frequency of the note. The function looks like this:

[2]boolean means its a value that is either true or false (0,1)

```
function y = prettyNote1(TheNote,bPlay,bDisplay)
if nargin <2,    bPlay = false;     end     %default values of inputs
if nargin <3,    bDisplay = false;  end
Fs = 40000;                                 %sampling frequency
P = TheNote(1);                             %the frequency of the note
L = TheNote(2);                             %the duration of the note
t  = linspace(0,L,Fs*L);                    %time interval

   %BEGIN COMPUTATION
y  = sin(t*P);                              %pure tone
   %END COMPUTATION

if bPlay
  sound(y,Fs);
  pause(L);
end
if bDisplay
  figure;
  plot(t,y);
  xlim([0   ,0.01]);   %x axis limit to 0.01 seconds
  ylim([-1.5,1.5]);
end
```

To try out our function, we can use it in a few different ways:

```
theNote  =  [4000,1];              %a note of 4k frequency and 1 sec duration
  prettyNote1(theNote,1  );        %play the note
  prettyNote1(theNote,0,1);        %display audio vector of the note
  prettyNote1(theNote,1,1);        %play and display
y=prettyNote1(theNote    );        %gather the audio vector into 'y'
```

2.1 Audio Attenuation

The tone doesn't sound very pleasant(pure tones aren't used much in music). To make a nicer sounding tone lets implement a gradual decrease in amplitude over time(sound attenuation). Mathematically:

$$y = \sin(tP) \times \text{lin}(t)$$

where $\text{lin}(t)$ is a function that linearly decreases from 1 to 0

In code, this can be done in two steps: first generate a linearly decreasing vector and then multiply it with the audio. It is implemented in prettyNote2:

```
   %from the function "prettyNote2.m"
   %BEGIN COMPUTATION
tone = sin(t*P);                              %pure tone
lin  = linspace(1,0,Fs*L);                    %linear decrease(attenuation)
y    = tone.*lin;                             %generate the audio
   %END COMPUTATION
```

Try running it with `prettyNote2(theNote,1,1);`
Using our function `prettyNote2`, we can now get some nicer sounding music:

```
Fs = 40000;                                   %sampling frequency
Bt = 6000;                                    %base tone
y1 = prettyNote2([Bt*1,     0.4 ]);
y2 = prettyNote2([Bt*1.115, 0.45]);
y3 = prettyNote2([Bt*0.89,  0.55]);
y4 = prettyNote2([Bt*0.45,  0.4 ]);
y5 = prettyNote2([Bt*0.67,  1   ]);

yTotal = [y1 y2 y3 y4 y5];                    %merge(concatenate) the vectors
sound(yTotal,Fs);                             %play the full song
```

2.2 Overtones

To improve the pleasantness further, lets add *overtones* to the audio. Here is a script that visualizes the first 2 overtones of a signal:

```
clear;
P = 1;
t = linspace(-pi,pi,5000);
plot(t, sin(t*P  ),   ...          %original signal
     t, sin(t*P*2),   ...          %first overtone...
     t, sin(t*P*3));               %second

ylim([-2,2]);
legend('original','OT 1','OT 2');
```

For our purposes overtones are defined as tones that have frequencies related to the original tone by integer multiplication[3]. The function `prettyNote3` adds one overtone:

```
%from the function "prettyNote3.m"
   %BEGIN COMPUTATION
lin  = linspace(1,0,length(t));       %linear decrease(attenuation)
y  =  0.7*sin(t*P  ).*lin + ...       %tone with linear attenuation
      0.3*sin(t*P*2).*lin.^2;         %1st overtone with attenuation
   %END COMPUTATION
```

Apart from just adding the overtone, we also put different attenuation on it (that equals $\text{lin}^2(t)$). To hear this sound, and get both tones(original and overtone) visualized try out:

```
prettyNote3([6000 2],1,1);
```

[3] the overtones we consider are so-called harmonic overtones.

A tone $\sin(tPk)$, for some positive integer k is an overtone of $\sin(tP)$, and could be attenuated by $\text{lin}^k(t)$ in our synthesizer. Mathematically, we can synthesize as follows:

$$y = \sum_{k=1}^{K} c_k \sin(tPk) \times \text{lin}^k(t)$$

The function `prettyNote3` above, does this for $c_1 = 0.7$, and $c_2 = 0.3$. A sum, of the kind in above equation, is easily implemented in matlab with a `for` loop. In `prettyNote4`, we gather the different weights ('c_k') in the vector `c` as we investigate below:

```
   %code from prettyNote4.m:
   %BEGIN COMPUTATION
c   = [ 0.6    0.1    0.3];
lin = linspace(1,0,length(t));              %linear decrease(attenuation)
y   = c(1)*sin(t*P).*lin;                   %the main tone

for k = 2:length(c)
  y = y + c(k)*sin(t*P*k).*lin.^k;          %summation of overtones
end
   %END COMPUTATION
```

Hopefully, the sound now somewhat resembles that of a plucked string.

Task 1

Make a new function `prettyNote4.m` which will synthesize sound, and share the same syntax as described in section 2. The function `prettyNote4` should add a so-called *tremolo* effect. Mathematically, we want to do the following:

$$y_{new} = y_{old} \times \frac{1 + \sin(t\frac{P}{90})}{2}$$

where y_{old} is the audio as generated in `prettyNote4`, i.e. with attenuation and overtones effects.

Simply put, tremolo is the effect of sound periodically increasing and decreasing in amplitude over time.Tremolo is often mistaken for vibrato, which is a rapid change in frequency, not amplitude. The idea is to make a secondary oscillating function: $\frac{1+\sin(tP/M)}{2}$, of much lower frequency than the pure tone.

Hints

The audio file 'e2output.wma' lets you hear how function `prettyNote4` is used in music (with some bird chirping as well).

3 Making music

In our examples of playing music so far, we have used single lines for both the definition and the generation of each note, such as:

```
Bt = 6000;
prettyNote4([Bt*0.45,  1 ],1);
```

In above example, we generate one note that is 0.45 frequency of the base tone, and is 1 second long. It is convenient to have all the notes and their durations listed in one place(this is what a sheet of musical notes are). This is an example of a signal that is $\mathbb{R}^1 \rightarrow \mathbb{R}^2$ the sheet:

```
Sheet = [1     1.115   0.89   0.45    0.67; ...
         0.4   0.45    0.55   0.4     1  ]
```

To play a sheet using our synthesizer, use playSheetOfMusic1.m that can read and play musical sheets:

```
function y = playSheetOfMusic1(bPlay,BaseTone,BaseBeat,Sheet)
if nargin < 1,  bPlay = true;    end
if nargin < 2,  BaseTone = 6000; end
if nargin < 3,  BaseBeat = 1;    end
if nargin < 4,
  Sheet = [1    1.115 0.899  0.45 0.675; ...
           0.4  0.45  0.55   0.4  1  ];
end
Fs = 40000;
Sheet(1,:) = Sheet(1,:)*BaseTone;          %scale the frequencies with base tone
Sheet(2,:) = Sheet(2,:)/BaseBeat;          %scale the durations with base beat
nofTones   = size(Sheet,2);                %nofTones = nofColumns in sheet

   %BEGIN AUDIO GENERATION
y = [];                              %y starts as the empty vector
for k = 1:nofTones
  aNote = Sheet(:,k);
  yNew = prettyNote4(aNote);
  y = [y yNew];                      %concatenate new audio at end of previous audio
end
   %END AUDIO GENERATION
if bPlay
   sound(y,Fs);                         %play the full song
   totDuration = sum(Sheet(2,:));
   pause(totDuration);
end
```

4 Musical Notes

A note is usually quantized into 7 characters of increasing pitch (A, B, C, D, E, F, G). Because this is rather coarse, there are notes that occupy pitches in-between, increasing the quantization. The pitch that is one half higher than C is called *C sharp*, and denoted $C\#$. To make it more complicated, *C sharp* is equivalent to *D flat*, denoted Db which is achieved by taking "one half lower" as seen in figure 1.

Figure 1: Mapping of Letter code to piano, with sharps(#) and flats(b)

And just to make it more complicated, some notes have no in-between (missing black keys). There is no black key for E#, but it is still defined in musical notation as equal to F (the closest valid higher tone, as seen in figure 1).

A good way to start keeping track of these different pitch-notations is by a Matlab ***cell-array***. A first single label on each key in figure 1 is given by the cell-array:

```
PitchNames  = ...
  {'C', 'C#', 'D', 'D#', 'E', 'F', 'F#', 'G', 'G#', 'A', 'A#', 'B'}
```

Every entry in `PitchNames` correspond to a specific frequency of audio. Below is a list of frequencies that works pretty ok for our purposes. It is naturally written like a vector, the length is the same as `PitchNames`:

```
PitchFreqs = ...
  [0.53, 0.56, 0.595, 0.63, 0.667, 0.707, 0.75, 0.794, 0.841, 0.891, 0.944, 1]
```

What we would like to have is a way to map the names of the pitches, to the specific frequencies they correspond to. We do this by a Hash Map:

```
% create a hash map using 'PitchNames' and 'PitchFreqs'
Fr = containers.Map(PitchNames, PitchFreqs);

% get some frequencies of notes:
Fr('C')
Fr('F#')

% error when using unregistered notation (E# was not in the 'PitchNames' list):
Fr('E#')
Fr('Eb')
```

The object Fr works as a translator, giving us a value for frequency, given that we have a character tone.

In musical notation, 'E#' and 'Eb' are valid notes (you can find them in musical sheets). By convention, 'E#' is equivalent to 'F' (its the next higher key in figure 1), and 'Eb' is equivalent to 'D#'. There are several such 'enharmonic equivalents' in modern notation. From figure 1, we have two sources of equivalents: **1) 'missing black keys'**, and **2) 'Redundancy of sharps and flats'**.

The equivalence relations from 'missing black keys' are:

- $B\# \equiv C$
- $Cb \equiv B$

- $E\# \equiv F$
- $Fb \equiv E$

The equivalence relations from 'Redundancy of sharps and flats' are:

- $Ab \equiv G\#$
- $Bb \equiv A\#$
- $Cb \equiv B\#$
- $Db \equiv C\#$
- $Eb \equiv D\#$
- $Fb \equiv E\#$
- $Gb \equiv F\#$

We can upgrade our hash map to reflect these equivalences by:

```
%from `missing black keys':
Fr('B#') = Fr('C');
Fr('Cb') = Fr('B');
Fr('E#') = Fr('F');
Fr('Fb') = Fr('E');

%from 'sharps and flats':
Fr('Ab') = Fr('G#');
Fr('Bb') = Fr('A#');
Fr('Cb') = Fr('B#');
Fr('Db') = Fr('C#');
Fr('Eb') = Fr('D#');
Fr('Fb') = Fr('E#');
Fr('Gb') = Fr('F#');
```

In the above snippet, the order is important. For each line, we make a new entry in the hash map using an existing entry. For example,

```
Fr('B#') = Fr('C');
```

must be executed before:

```
Fr('Cb') = Fr('B#');
```

The values from `Fr` (those originally in `PitchFreqs`) are bounded in (0,1) and contain no information on duration. We require a base tone and a duration of the note to play it:

```
% Play a 'E#' with base tone 6k, for 1 second duration
Bt = 6000;
aNote = [Bt*Fr('E#'), 1];
prettyNote4(aNote,1);
```

5 Composing Music

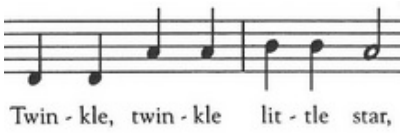

Figure 2: A very recognizeable tune :)

And, so here we are. We have made our own synthesized audio, our own interpreter for notes, and a way to make some music from scratch. Lets do some composing, start by implementing the musical sheet of figure 2:

```
Sheet = ...
[Fr('C'), Fr('C'), Fr('G'), Fr('G'), Fr('A'), Fr('A'), Fr('G'),
 1      , 1      , 1      , 1      , 1      , 1      ,  2     ];
playSheetOfMusic1(1,6500,2,Sheet);
```

In above snippet (as the previous ones) we are listing all the frequencies (first row), then all the durations(second row). A fully equivalent way (identical to previous snippet) is to write the `Sheet` transposed, and then transpose it back:

```
Sheet = ...
[Fr('C'),1; ...
 Fr('C'),1; ...
 Fr('G'),1; ...
 Fr('G'),1; ...
 Fr('A'),1; ...
 Fr('A'),1; ...
 Fr('G'),2];

Sheet = transpose(Sheet);

playSheetOfMusic1(1,6500,2,Sheet);
```

Lets do a bit more of the same song, but we increase the base beat, and reduce the base tone:

```
Sheet = ...
[Fr('C'),1; Fr('C'),1; Fr('G'),1; Fr('G'),1; Fr('A'),1; Fr('A'),1; Fr('G'),2; ...
 Fr('F'),1; Fr('F'),1; Fr('E'),1; Fr('E'),1; Fr('D'),1; Fr('D'),1; Fr('C'),2]';

playSheetOfMusic1(1,4500,2.5,Sheet);
```

A small modification of the end tone yields something new:

```
Sheet = ...
[Fr('C'),1;Fr('C'),1;Fr('G'),1;Fr('G'),1;Fr('A'),1;Fr('A'),1;Fr('G'),2;...
 Fr('F'),1;Fr('F'),1;Fr('E'),1;Fr('E'),1;Fr('D'),1;Fr('D'),1;...
 Fr('E'),.4;Fr('C'),1.6]';

playSheetOfMusic1(1,4500,2.5,Sheet);
```

... and finally, how about this cute version:

```
Sheet= ...
[Fr('C'),1  ; Fr('C'),1  ; Fr('G'),1  ; Fr('G'),1; ...
 Fr('A'),0.8; Fr('C'),0.3; Fr('A'),1  ; Fr('G'),2; ...
 Fr('F'),0.8; Fr('D'),0.3; Fr('F'),1  ; Fr('E'),1; ...
 Fr('E'),1  ; Fr('D'),0.8; Fr('F'),0.3; Fr('D'),1; Fr('C'),2]';

playSheetOfMusic1(1,5500,3.5,Sheet);
```

6 Audio Recognition

At the very coarsest scale of understanding, any human listener would say we are dealing with the same song in the above examples. To make such a decision based solely on the audio signals sample values seems absurd. More reasonable is to divide this problem into scales of interest. From the audio samples, we may want to infer the tones, and at the same time the effects and frequency contents of those tones. On a higher level still, we have segmented out the different tones, and on the final, coarsest scale we may want to match towards known songs, and would need the equivalence relations introduced.
These are very difficult problems, the topics of many past and future phd student projects.

Task 2

(Some additions to this task will be made very shortly, you will get some more hints and some additional code to work with. If you are brave, try to solve it without those extra hints).
Make a script which will try to determine the musical sheet of synthesized sound. Use the `twinkleMadness` script as a starting point. The script should first make synthesized music, based on the twinkle little star song. After the audio has been generated, it should be analyzed based on a Gabor filter bank. Have one Gabor filter for every tone you wish to detect, let the Gabor filter with highest magnitude response indicate the detected tone.

The first thing you need to recover is a signal that is much coarser than the original, 10 hz will work well.

This signal will contain the average Gabor filter responses. If you have N filters, this signal will be $\mathbb{R}^1 \rightarrow \mathbb{R}^N$

Make a desciscion for every such sample of what tone is being played.

Run your system first on `PrettyNote1` synthesized audio only, and then on the more complicated prettyNote functions. Try to see what happens if you put some extra noisy signal in the background.